Asymptotic notation and recursion

Code: https://github.com/mesmere/sasha-tutorial/tree/main/2024-01-27

Asymptotic notation

Asymptotic notation

- We're interested in comparing the **long-run behavior** of functions.
- "Asymptotic" = we do not care what happens for small values.
- If, in a certain mathematical sense, a function g(x) **dominates** another function f(x) **in the long run**, we say that:

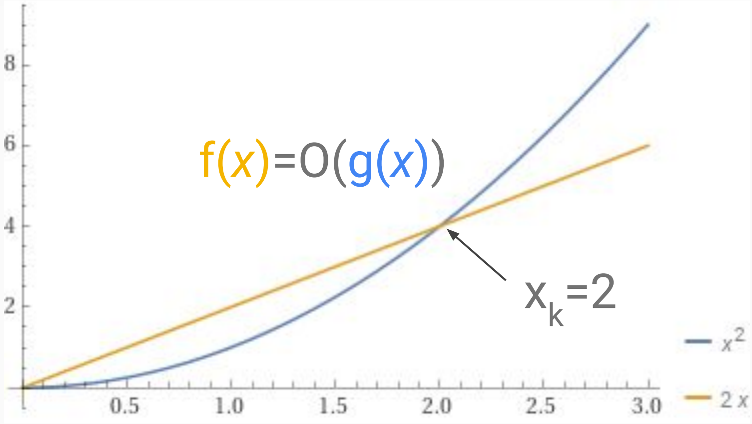
$$f(x)=O(g(x)).$$

Big-O

Mathematical definition:

$f(x)=O(g(x))$ iff.

$$\exists\, b, x_k \;\; \forall x > x_k : b \cdot g(x) > f(x)$$



Plain English:

$f(x)$ is big-O of $g(x)$ if and only if:

there is some point x_k **beyond which** $g(x)$ always dominates $f(x)$

(we're allowed to scale $g(x)$ by some constant factor b to make it work)

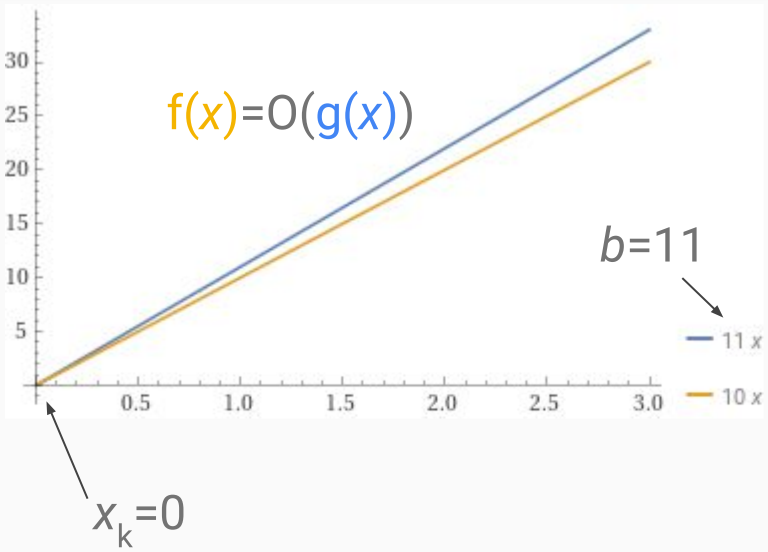
Big-O examples - constant factors

Constant factors don't matter…

f(x)=10x, g(x)=x

Is 10x=O(x)? YES. We can pick b=11 as our scaling factor to dominate 10x.

So we say that "10x is big-O of x" or "**linear in x**" or "order x."



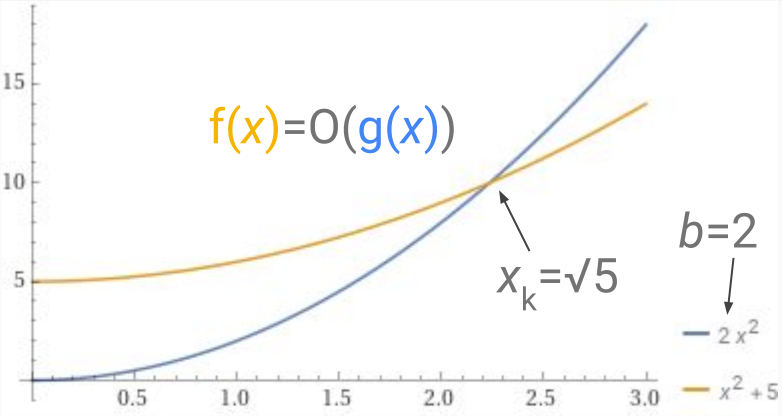
Big-O examples - constant offsets

Constant offsets don't matter either…

f(x)=x^2+5, g(x)=x^2

Is $x^2+5=O(x^2)$? YES. Even a small scaling factor like b=2 will dominate any constant offset in the long run.

So we say that "x^2+5 is O(x^2)" or "**quadratic in x**" or "order x squared."



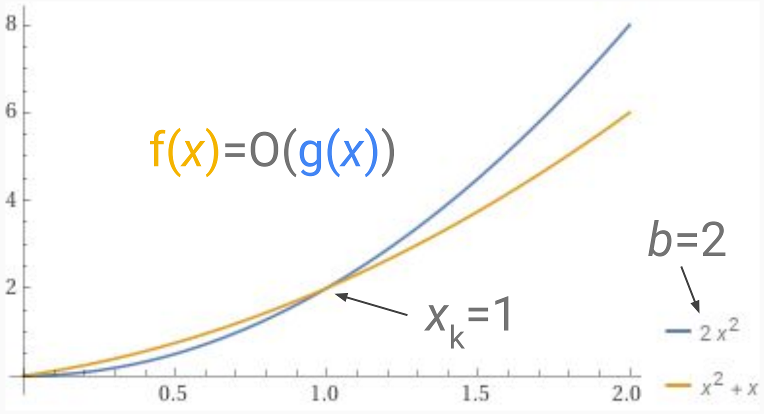
Big-O examples - lower-order terms

We can always ignore lower-order terms…

$f(x)=x^2+x$, $g(x)=x^2$

Is $x^2+x=O(x^2)$? YES. In a polynomial, scaling the highest-order term dominates lower-order terms in the long run. Pick $b=2$.

So we say that "x^2+x is $O(x^2)$" or "**quadratic in x**" or "order x squared."



Big-O examples - logs

$a=2^c \Leftrightarrow c=\lg a$

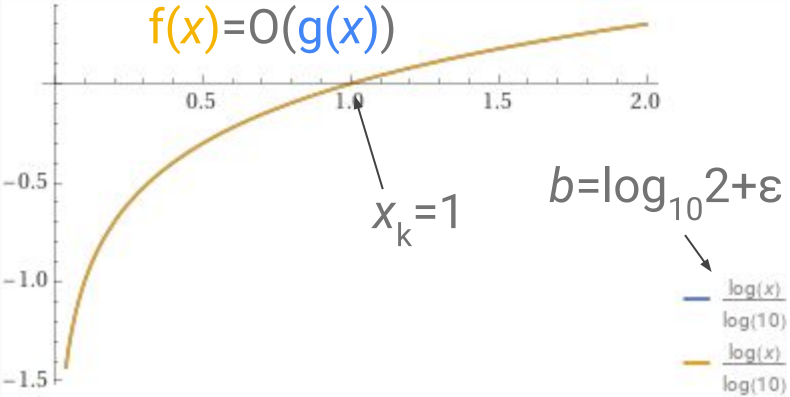
c is the “number of doublings” required to get from 1 to a.

Log bases might as well all be 2…

$f(x)=\log_{10}x$, $g(x)=\lg x$

Is $\log_{10}x=O(\lg x)$? YES. By the log change of base formula, $\log_{10}x = (\lg x)\cdot(\log_{10}2)$ and this is just multiplying by a constant factor!

So we say that “$\log_{10}x$ is $O(\lg x)$” or “order $\lg x$.”



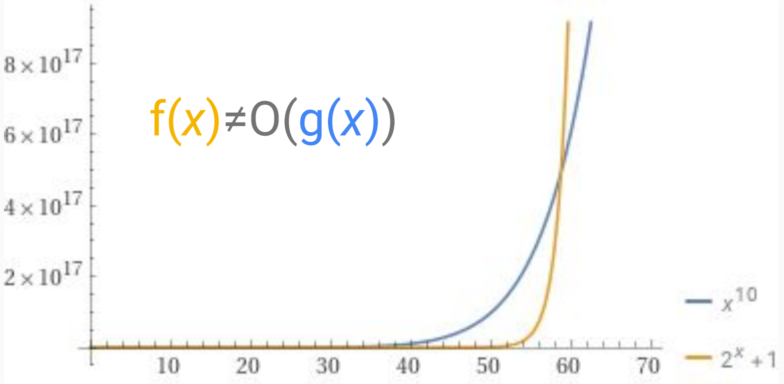
Big-O examples - exponentials

Exponentials are strong…

$f(x)=2^x+1$, $g(x)=x^{10}$

Is $2^x+1=O(x^{10})$? NO. Check out the behavior long-term. Exponentials win in the end, and no amount of scaling will help.

We say instead that "2^x+1 is $O(2^x)$" or "**exponential in x**" or "order two to the *x*."



Big-O review

- Constant factors don't matter.
- Constant offsets don't matter.
- Lower-order terms don't matter.
- Log bases don't matter.
- Exponentials (c^x) dominate polynomials (x^c).

Common comparisons:

$$O(1) < O(\lg n) < O(n) < O(n \lg n) < O(n\text{^}2) < O(n\text{^}3) < \ldots < O(2\text{^}n) < O(3\text{^}n) < \ldots < O(n!)$$

constant | log | linear | quadratic | factorial

polynomial | exponential

Big-theta

$f(x)=\Theta(g(x))$

Alternate notation:
$f(x) \asymp g(x)$

So far we've only cared whether one function will dominate another.

Example: $2x^2 = O(x^3)$.

But it's often useful to bound both above and below with different *b*. This is what Θ does.

Example:
$2x^2 \neq \Theta(x)$
$2x^2 \neq \Theta(x^3)$
$2x^2 = \Theta(x^2)$

In practice people say "big O" even when they technically mean "big theta."

What does this have to do with code?

We can characterize each algorithm's costs like running time, memory usage, network requests, etc. as **a function of the problem size**. Then we can asymptotically compare two algorithms to see which one will be more efficient for sufficiently-large problems.

What is f(x)? The cost we want to measure, expressed in terms of some x.

Sort an array: x is the size of the array, f(x) is the number of comparisons.

Multiply numbers: x is the number of bits required to represent the input, f(x) is the number of math operations required to get the answer.

Graph algorithms: Often multivariate, e.g. n is node count, m is edge count.

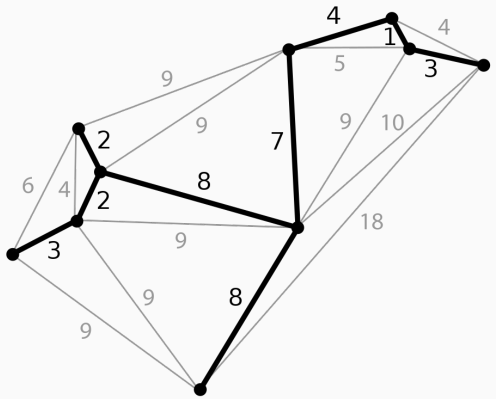
Asymptotic performance isn't everything

Beware:

- Constants may be large in practice, but we ignore them!
- This means that x_k may need to be absurdly gigantic for a theoretical improvement to ever pay off.

E.g. "greedy algorithm" for finding a **minimum spanning tree**:

```
Mark a random node.
While any unmarked nodes remain:
  Look at all edges from marked nodes to
  unmarked nodes.
  Add the smallest such edge to our MST.
  Mark the node on the other side.
```



Exercise 1 - Linear search

```
function search(arr, value) {
  for (const cur of arr) {
    if (cur === value) {
      return true;
    }
  }
  return false;
}
```

How many comparisons does this take, as a function of `arr.length`?

Exercise 2 - Selection sort

```
function selectionSort(arr) {
  // Grow the sorted range by 1 with each iteration.
  for (let sortedUpToIdx = 0; sortedUpToIdx < arr.length - 1; sortedUpToIdx++) {
    // Find the smallest item in the remaining unsorted range.
    let smallestSoFarIdx = sortedUpToIdx;
    for (let candidateIdx = smallestSoFarIdx + 1; candidateIdx < arr.length; candidateIdx++) {
      if (arr[candidateIdx] < arr[smallestSoFarIdx]) {
        smallestSoFarIdx = candidateIdx;
      }
    }

    // Swap the smallest remaining item to the end of the sorted range.
    const oldOccupant = arr[sortedUpToIdx];
    arr[sortedUpToIdx] = arr[smallestSoFarIdx];
    arr[smallestSoFarIdx] = oldOccupant;
  }
}
```

How many comparisons does this take, as a function of `arr.length`?

Recursion

How does JS keep track of what's executing?

Calling a function **pushes** a new frame onto the top of the call stack
and begins execution in the new frame.
Returning from a function **pops** the top frame off of the call stack
and resumes execution in whatever frame is at the top next.

```
fun1();

function fun1() {
  fun2();
}

function fun2() {
  fun3();
}

function fun3() {
  console.trace();
}

```

```
Trace
    at fun3 (solution.js:12:11)
    at fun2 (solution.js:8:3)
    at fun1 (solution.js:4:3)
    at solution.js:1:1
```

Recursion example - factorial

Definition: $n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n$

$5! \quad = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5$

$\quad\;\; = 4! \cdot 5$

$4! \quad = 1 \cdot 2 \cdot 3 \cdot 4$

$\quad\;\; = 3! \cdot 4$

General rule:

$n! \quad = (n - 1)! \cdot n \qquad \text{when } n > 0$

$n! \quad = 0 \qquad \text{when } n = 0$

```
function fact(n) {
  if (n == 0) {
    return 1; // Base case
  }

  return n * fact(n-1); // Recursion
}
```

Recursion example - factorial (cont.)

```javascript
function fact(n) {
  console.log(`Evaluating fact(${n})...`);

  if (n == 0) {
    console.log("Base case - returning 1.")
    return 1;
  }

  console.log(`Recursion case - evaluating ${n} * fact(${n-1}).`)
  const answer = n * fact(n-1);
  console.log(`Returning ${answer} to the caller.`)
  return answer;
}

fact(5);
```

```
Evaluating fact(5)...
Recursion case - evaluating 5 * fact(4).
Evaluating fact(4)...
Recursion case - evaluating 4 * fact(3).
Evaluating fact(3)...
Recursion case - evaluating 3 * fact(2).
Evaluating fact(2)...
Recursion case - evaluating 2 * fact(1).
Evaluating fact(1)...
Recursion case - evaluating 1 * fact(0).
Evaluating fact(0)...
Base case - returning 1.
Returning 1 to the caller.
Returning 2 to the caller.
Returning 6 to the caller.
Returning 24 to the caller.
Returning 120 to the caller.
```

Recursion example - Fibonacci sequence

The sequence which begins: 1, 1, 2, 3, 5, 8, 13, 21...

Definition:

fib(n) = fib(n - 1) + fib(n - 2) when $n \geq 2$

fib(n) = 1 when n = 0, 1

```
function fib(n) {
  if (n == 0 || n == 1) {
    return 1;
  }

  return fib(n-2) + fib(n-1);
}
```

Recursion example - Fibonacci (cont.)

fib(10) = fib(8) + fib(9)

= fib(6) + fib(7) + fib(7) + fib(8)

= fib(4)+fib(5)+fib(5)+fib(6) + fib(5)+fib(6)+fib(6)+fib(7)

= …

= fib(0)+fib(1)+fib(1)+fib(0)+fib(1)+ … +fib(0)+fib(1)+fib(1)+fib(0)+fib(1)

Quiz: How many leaves are on this tree?

Hint: fib(0)=fib(1)=?

Recursion example - Fibonacci (cont.)

Math knowledge:

> fib(n) is a linear recurrence relation. Solving, you find that fib(n)$\propto\varphi^n$ where φ is a constant (the golden ratio).

CS knowledge:

> The recursion tree has a constant branching factor so its total number of nodes is O(its number of leaves).

Analysis: our fib(n) runs in O(φ^n), a.k.a. "exponential time."

Recursion example - Fibonacci (cont.)

fib(10) = fib(8) + fib(9)

= fib(6) + fib(7) + fib(7) + fib(8)

= fib(4)+fib(5)+fib(5)+fib(6) + fib(5)+fib(6)+fib(6)+fib(7)

= …

= fib(0)+fib(1)+fib(1)+fib(0)+fib(1)+ … +fib(0)+fib(1)+fib(1)+fib(0)+fib(1)

Look at all of these overlapping subproblems!

Recursion example - Fibonacci (cont.)

Memoization - cache the results of recursive calls.

```
function fib(n) {
  if (n == 0 || n == 1) {
    return 1;
  }

  return fib(n-2) + fib(n-1);
}
```

$\Theta(\varphi^n)$ calls $\longrightarrow$ $\Theta(n)$ calls

```
function Fibber() {
  this.cache = {};
  this.fib = function (n) {
    if (n == 0 || n == 1) {
      return 1;
    }

    if (this.cache[n] === undefined) {
      this.cache[n] = this.fib(n-2) + this.fib(n-1);
    }
    return this.cache[n];
  }
}

new Fibber().fib(5);
```

Recursion example - Binary search

Problem:

Given a number *n* and a sorted array of numbers, is *n* in the array?

Naive solution:

Check the entire array from left to right until you find *n* or reach the end.

Quiz: What is the average-case performance of our naive solution?

…we can do better with binary search…

Recursion example - Binary search (cont.)

Start in the middle and either go left or right depending on whether the element you're looking for is less or greater than the element in the middle.

```javascript
function binarySearch(ary, needle) {
  // Base case - no elements
  if (ary.length === 0) {
    return false;
  }

  // Base case - one element
  if (ary.length === 1) {
    return needle == ary[0];
  }

  // Recursion case - multiple elements
  const midpoint = Math.floor(ary.length/2);
  if (needle < ary[midpoint]) {
    return binarySearch(ary.slice(0, midpoint), needle);
  } else {
    return binarySearch(ary.slice(midpoint, ary.length), needle);
  }
}
```

Example:
ary = [2, 3, 5, 7, 11, 13, 17]

Compare with `needle`.

Recursion example - Binary search (cont.)

```
function binarySearch(ary, needle) {
  console.log(`Searching ${ary} for ${needle}`);

  // Base case - no elements
  if (ary.length === 0) {
    return false;
  }

  // Base case - one element
  if (ary.length === 1) {
    return needle == ary[0];
  }

  // Recursion case - multiple elements
  const midpoint = Math.floor(ary.length/2);
  if (needle < ary[midpoint]) {
    return binarySearch(ary.slice(0, midpoint), needle);
  } else {
    return binarySearch(ary.slice(midpoint, ary.length), needle);
  }
}

binarySearch([2, 3, 5, 7, 11, 13, 17], 15);
```

Example execution...

```
Guest ran 23 lines of JavaScript (finished in 532ms):

Searching 2,3,5,7,11,13,17 for 15
Searching 7,11,13,17 for 15
Searching 13,17 for 15
Searching 13 for 15
>
```

Recursion example - Binary search (cont.)

Quiz: How many comparisons will `binarySearch` perform in the worst case, as a function of the input array length *n*?

Hint: We're starting with *n* elements and cutting the array in half repeatedly until we get to 1 element. How many halvings does it take to get from *n* down to 1?

Hint 2: It's the same number of doublings it takes to get from 1 up to *n*. 🤔

Divide and conquer

Procedure:

1. Divide the problem into pieces.
2. Solve the pieces independently. **(Recursion!)**
3. Combine the answers to get an overall answer.

Divide and conquer example - Merge sort

Problem:

Given an unsorted array of numbers, sort it.

Solution:

1. Split the array down the middle.
2. Sort the first half and the second half independently. **(Recursion!)**
3. Merge the two sorted arrays into a single sorted array.

Divide and conquer example - Merge sort

```
function mergeSort(ary) {
  // Base case - zero or one elements
  if (ary.length <= 1) {
    return ary;
  }

  // Recursively sort each half.
  const midpoint = Math.floor(ary.length / 2);
  const left = mergeSort(ary.slice(0, midpoint));
  const right = mergeSort(ary.slice(midpoint, ary.length));

  // Merge the two sorted halves into a single sorted array.
  let leftIdx = 0, rightIdx = 0;
  let result = [];
  while (leftIdx + rightIdx < ary.length) {
    if (leftIdx === left.length) {                    // If we're out of elements on the left...
      result.push(right[rightIdx++]);                 // ...take from the right.
    } else if (rightIdx == right.length) {            // If we're out of elements on the right...
      result.push(left[leftIdx++]);                   // ...take from the left.
    } else if (left[leftIdx] <= right[rightIdx]) {    // If next in line on left <= in right...
      result.push(left[leftIdx++]);                   // ...take from the left.
    } else {                                          // If next in line on left > on right...
      result.push(right[rightIdx++]);                 // ...take from the right.
    }
  }
  return result;
}
```

Divide and conquer example - Merge sort

```
mergeSort([3, 2, 1, 4])  ⇒  merge(mergeSort([3, 2]), mergeSort([1, 4]))

mergeSort([3, 2])        ⇒  merge(mergeSort([3]), mergeSort([2]))
mergeSort([3])           ⇒  [3]   // Base case
mergeSort([2])           ⇒  [2]   // Base case

mergeSort([1, 4])        ⇒  merge(mergeSort([1]), mergeSort([4]))
mergeSort([1])           ⇒  [1]   // Base case
mergeSort([2])           ⇒  [2]   // Base case
```

Divide and conquer example - Merge sort

What is the **runtime performance** of merge sort?

- Often for a comparison sort like merge sort, we're interested in counting the worst-case number of comparisons between array elements.
- Does the answer change if you count "CPU instructions" instead?